# FACILITATING REPRODUCIBLE COMPUTING VIA SCIENTIFIC WORKFLOWS -- AN INTEGRATED SYSTEM APPROACH

by

**Yuan Cao**

**A Thesis**

*Submitted to the Faculty of Purdue University*

*In Partial Fulfillment of the Requirements for the degree of*

**Master of Science**


Department of Computer Sciences

Indianapolis, Indiana

May 2017

# THE PURDUE UNIVERSITY GRADUATE SCHOOL
# STATEMENT OF THESIS APPROVAL

Dr. Yao Liang, Chair

Department of Computer and Information Science

Dr. Fengguang Song

Department of Computer and Information Science

Dr. Jiang Yu Zheng

Department of Computer and Information Science

**Approved by:**

Dr. Shiaofen Fang

Head of the Departmental Graduate Program

# ACKNOWLEDGMENTS

This thesis would not be done without the help and support of many people.

I would like to express my gratitude to my adviser Prof. Yao Liang for his great support and help as I worked on graduate courses and the thesis. His supervision helped expedite my research progresses and opened the door to new discoveries.

I would also like to thank my committee members Prof. Fengguang Song and Prof. Jiang Yu Zheng for their time and guidance. The Department of Computer & Information Science, Purdue University at Indianapolis has provided an excellent environment for my study and research.

In addition, I would like to thank Prof. Shiaofen Fang, Prof. James Hill, Prof. Mohammad Al Hasan, Prof. John Gersting and Prof. Rajeev R. Raje for their wonderful courses. I learned a lot from these inspiring classes, and have applied what I gained in these classes to my research work.

I am also thankful to many department staff, including, but not limited to, Joan, Nicole and Scott and all people and students, especially Yimei Li, Xiaoyang Zhong, from my department, for their patience and help as they came along with me during this process.

Finally, I would like to thank to my parents and my wife for their love and support.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# ABBREVIATIONS

| | |
|---|---|
| API | Application Program Interface |
| COM | Component Object Model |
| DLL | Dynamic Link Library |
| GUI | Graphical User Interface |
| SWfMS | Scientific Workflows Management System |
| XML | Extensible Markup Language |

# ABSTRACT

Author: Cao, Yuan. MS
Institution: Purdue University
Degree Received: May 2017
Title: Facilitating Reproducible Computing via Scientific Workflows -- An Integrated System Approach
Major Professor: Yao Liang

Reproducible computing and research are of great importance for scientific investigation in any discipline. This thesis presents a general approach to provenance in the context of workflows for widely used script languages. Our solution is based on system integration, and is demonstrated by integrating MATLAB with VisTrails, an open source scientific workflow system. The integrated VisTrails-MATLAB system supports reproducible computing with truly prospective and retrospective provenance at multiple granularity levels as scientists choose for their scripts, and at the same time, is very easy to use.

# CHAPTER 1. INTRODUCTION

## 1.1. Research Background

In the era of today's computing and information technology, reproducible computing and research are of great importance for scientific investigation in any discipline, from broad computational sciences (e.g., physics, chemistry, biology, and neuroscience), to various computational engineering and modelings [1, 2, 3]. Taking an example of the study of climate change, scientists need to simulate complex physical phenomena and behaviors based on various processes including climate, hydrological, environmental, chemical, ecological, and biological, under various conditions and parameters, where computational models are developed for individual processes, and different models are coupled with each other to account for the complicated interactions among individual physical processes. Each involved individual model is realized in a software module or system, thus the overall coupling modeling forms a complex modeling system of systems.

To investigate such a modeling system of systems is not an easy task. Its complexity comes from the following aspects:

(1) A number of individual models to select and different ways to form a computing workflow
(2) Different input and output structures and formats of each model
(3) Various data sets from different data sources to select (and to fuse)
(4) Potentially a huge parameter configuration space of models to explore

Therefore, it is not surprising to see that it is very challenging for one group or organization's published modeling research results to be reproduced by another research group or organization, which makes the verification, validation, evaluation and sharing of any new modeling result very difficult in scientific communities.

In view of the above problems, scientific workflows with provenance are an effective tool to address the reproducibility issues [e.g., 4, 5, 6]. A computing pipeline of diverse model coupling can be accurately described as a scientific workflow, in which each model can be expressed as a workflow item. Depending on the level of granularity

at which data dependency needs to be observed, a model itself can be further described as a subworkflow composed of multiple processing steps. With an appropriate scientific workflows management system (SWfMS) framework, the execution of a constructed workflow with selected original data sets under given parameter configuration can be precisely recorded through SWfMS' (retrospective) provenance functionality at run time. The recorded provenance information can be used later to trace the lineage of a particular result by identifying its corresponding data inputs, parameter setting, and the processing steps and the model couplings used to produce it, enabling the full reproducibility of the entire complex computational experiment/modeling process.

On the other hand, scripting languages, such as Python, R, and MATLAB, have been widely used in various scientific disciplines for computational experiments, engineering modelings, and data analytics. During the life cycle of scientific experiments, scientists compose scripts, execute them, and conduct analysis on the results. Then, depending on the analysis on the obtained results, they modify their scripts, input data sets, and/or parameter setting, to get more results and refine the experiment. This exploratory process continues until some satisfactory research goal is achieved. However, native Python, R and MATLAB scripts do not support scientific workflow and provenance, making reproducibility a great challenge to scientists in the scientific community. To address this issue, a recent new software tool noWorkflow [7] makes use of Python runtime profiling functions to reveal provenance traces of the execution of the script, providing retrospective provenance information similar to that in SWfMS for Python users. A drawback of noWorkflow is that its provenance information is captured at the code level of script statement, instead of at scientific workflow level. As a result, the recorded provenance information is potentially overwhelming and not easy to use. Another new tool called YesWorkflow [8] produces prospective provenance in script, which is achieved via first marking up scripts using a keyword-based annotation mechanism, in terms of comments of the host language, and then interpreting the structured annotation comments. However, the YesWorkflow has no retrospective provenance. McPhillips et al. extended YesWorkflow tool to infer retrospective provenance information [9]. Since the YesWorkflow tool and its extension are not invoked at all while a script is executing, a fundamental limitation of [9] is that no

dependencies between data inputs, parameter settings, intermediate results, and final outputs can be directly observed or recorded at run time. In other words, the retrospective provenance information provided in this way is inferred based on metadata but not observed from the actual execution of the script, and thus may not reflect the real retrospective provenance.

## 1.2. Definitions

### 1.2.1. What is Provenance?

Within computer science, informatics uses the term 'provenance' to mean the lineage of data, as per data provenance, with research in the last decade extending the conceptual model of causality and relation to include processes that act on data and agents that are responsible for those processes. [4]

With provenance scientist can interpret and understand a result, understand the experiment and chain of reasoning that was used in the production of a result, verify that an experiment was performed according to acceptable procedures, identify the inputs to an experiment were and where they came from, and track who performed an experiment and who is responsible for its results. In a sense, provenance is as important as results.

### 1.2.2. What is Scientific Workflow System?

Scientific workflow is an application comprising many tasks coupled by disk resident datasets. A scientific workflow system is a specialized form of a workflow management system designed specifically to compose and execute a series of computational or data manipulation steps, or workflow, in a scientific application.

## 1.3. Research Goal

In this thesis, our main goal is to present an alternative approach to retrospective provenance and reproducible computing in scripts. Our approach aims to provide a systematic framework to integrate any scripting language engine with an existing scientific workflow system such as VisTrails, so that scientists can easily wrap their scripts into scientific workflows with the scientific workflow system to take all the

advantages of the available SWfMS for truly prospective and retrospective provenance to achieve reproducible computing and research. While we use MATLAB in this paper to demonstrate our integrated system approach with VisTrails, our approach is general and can work with any scripting language engine.

### 1.4. Organization of the thesis

The rest of the thesis is organized as follows. Chapter 2 gives related works in scientific workflow management systems. Chapter 3 will presents our integrated system approach, the system design and architecture. Chapter 4 describes the system implementation. Chapter 5 provides a comprehensive use case for illustration. Finally, the conclusions and future work are given in Chapter 6.

# CHAPTER 2. RELATED WORK

Provenance is a critical concept in scientific workflows. It allows scientists to understand the origin of their results, to repeat their experiments, and to validate the processes that were used to derive data products.

## 2.1. noWorkflow

### 2.1.1. noWorkflow Overview

The full name of noWorkflow is not only workflow. It was presented by Leonardo Murta, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire in 2014 [7]. While scripts are widely used for data analysis and exploration in the scientific community, there has been little effort to provide systematic and transparent provenance management support for them. noWorkflow is a tool that transparently captures provenance of scripts and enables reproducibility. It can control flow information and library dependencies. It is non-intrusive and does not require users to change the way they work. Users do not need to wrap their experiments in scientific workflow systems, install version control systems, or instrument their scripts. The tool is non-intrusive and relies on techniques from Software Engineering, including abstract syntax tree analysis, reflection, and profiling, to collect different types of provenance without requiring a version control system or an instrumented environment. [7]

### 2.1.2. Drawbacks in noWorkflow

Although noWorkflow has its features, its drawbacks are also clear.

Firstly, noWorkflow's provenance information is captured at the code level of script statement, instead of at scientific workflow level. As a result, the recorded provenance information is potentially overwhelming and not easy to use.

Secondly, the noWorkflow provenance approach uses additional runtime provenance. It is typically at a much finer level of granularity. Runtime provenance recording may introduce significant execution overhead when not used with caution. This

overhead is often described in terms of the computation required to record the large numbers of events that occur during script execution. [9]

## 2.2. YesWorkflow

### 2.2.1. YesWorkflow Overview

Another new tool called YesWorkflow which complements noWorkflow by revealing prospective provenance in scripts. YesWorkflow produces prospective provenance in script, which is achieved via first marking up scripts using a keyword-based annotation mechanism, in terms of comments of the host language, and then interpreting the structured annotation comments. YesWorkflow requires neither the use of a workflow engine nor the overhead of adapting code to run effectively in such a system. Instead, YesWorkflow enables scientists to annotate existing scripts with special comments that reveal the computational modules and dataflows otherwise implicit in these scripts. YesWorkflow tools extract and analyze these comments, represent the scripts in terms of entities based on the typical scientific workflow model, and provide graphical renderings of this workflow-like view of the scripts.

### 2.2.2. Drawbacks of YesWorkflow

However, the YesWorkflow has no retrospective provenance. McPhillips et al. extended YesWorkflow tool to infer retrospective provenance information [9]. Since the YesWorkflow tool and its extension are not invoked at all while a script is executing, a fundamental limitation of retrospective provenance is that no dependencies between data inputs, parameter settings, intermediate results, and final outputs can be directly observed or recorded at run time. In other words, the retrospective provenance information provided in this way is inferred based on metadata but not observed from the actual execution of the script, and thus may not reflect the real retrospective provenance.

## 2.3. VisTrails

### 2.3.1. VisTrails Overview

VisTrails is an open-source SWfMS written in Python that supports for runtime provenance, visualization and data exploration. Users can contribute to VisTrails SWfMS by sharing bug reports, bug fixes, and suggestions with the VisTrails community. As scientists and engineers generate and evaluate various hypotheses and designs for their computational experiments, modeling simulations, data analysis and visualization, a series of different, albeit related, workflows are created through adjustment and modification of previous workflow(s) in an interactive process.

VisTrails SWfMS was designed to manage rapidly-evolving workflows, whose unique feature is a systematic provenance mechanism to record detailed history information including workflow steps and data inputted and derived during the execution of an exploratory task. This provenance information is persisted as XML files (or in a relational database). The provenance information allows users to navigate workflow versions in an intuitive way. It is also allowing users to analyze and visually compare different workflows and their derived results with different input data and parameter settings, and to support producible computing and research. A series operations and user interfaces are enabled in VisTrails that simplify workflow design and use. These operations and interfaces include the ability to create workflows, refine workflows by analogy and query workflows [10].

### 2.3.2. VisTrails' Important Features

The VisTrails SWfMS has many features:

- Flexible Provenance Structure.

VisTrails SWfMS can track all the changes which was made in workflows. These changes including all the explore steps. The VisTrails system can track the execution information in run-time of workflows. This information including who executed a module, elapsed time, on which machine and so on. VisTrails SWfMS also offers a flexible annotation structure, with this structure users can specify provenance information.

- Version Tree for Querying and Re-using.

The version tree stores all the provenance information which is a tree structure. Users can choose a relational database, such as MySQL, or XML files in the file system, to store this tree information. The VisTrails SWfMS provides flexible query interfaces through which users can reuse and explore workflow and provenance information.

- Easier for Collaborative Exploration.

The VisTrails SWfMS can be configured with a database backend. This database backend is being used as a repository which can be shared between different users. Multiple users can collaboratively explore through the synchronization mechanism.

- Extensibility.

VisTrails SWfMS provides a plugin functionality which is very simple and easy to use. Each user can dynamically add libraries and user packages. These extensions neither re-compilation of the system nor changes the user interfaces. Since VisTrails is written in Python, all of Python-wrapped libraries and third-party packages are easy to be integrated in this system.

- Parameter Exploration and Scalable Derivation of Data Products.

A series of operations which is for the simultaneous generation of multiple data products are supported in VisTrails SWfMS. These series of operations implemented by an interface that allows user to specify the value sets for different parameters in a workflow. VisTrails SWfMS contains a Spreadsheet, which display all the results of a parameter exploration side by side for easy comparison.

- Analogy Task Creation.

Analogies are first-class operations which is supported in VisTrails SWfMS. They are used to guide semi-automatic changes to multiple workflows. Users are not required to edit or manipulate the specifications of scientific workflow directly. [10]

# CHAPTER 3. APPROACH AND ARCHITECTURE

## 3.1. Design Objectives

We present the following objectives that guided the development of our reproducible computing approach.

- Objective 1: Provide scientists with a truly scientific workflow and provenance tool in their exploratory research with their favorable script languages such as MATLAB.

No scientific workflow and data provenance are supported in MATLAB. As MATLAB is commercial and closed source software, users have no way to modify and extend the MATLAB engine to add any runtime provenance mechanism. Therefore, it would be desirable to provide a framework enabling scientific workflow and provenance for MATLAB scripts for scientific communities.

- Objective 2: Support scientific workflow for script languages such as MATLAB at multiple granularity levels as scientists choose for their scripts.

Depending on the exploratory task at hand, scientists may need the very fine-gained provenance at statement level, or a coarse-grained provenance at module level (e.g., hundreds or thousands of lines of scripts), or multiple granularity levels simultaneously. A fine-gained provenance maintains very detailed execution history information for analysis but could be easily overwhelming. On the other hand, a coarse-grained provenance can be more efficient in many cases. Consequently, an effective workflow and provenance tool should support multiple granularity levels on demand.

- Objective 3: Support both prospective and retrospective provenance. In particular, support a truly retrospective provenance by recording provenance information at run time.

The prospective provenance of a script generates a workflow to explicitly reveal the structures of computational modules and dataflows that may be implicit in the script. On the other hand, retrospective provenance is the steps executed at the run time. Here, the ability to record provenance trace information during the execution is the key in

retrospective provenance, which requires building an underlying runtime recording system. In contrast, a recent work [9] on the extension of YesWorkflow tool can only infer rather than record retrospective provenance information at run time.

We propose to integrate a scripting language engine with VisTrails, by which the resulting integrated system makes good use of VisTrails' scientific workflow and provenance for the scripts. Scientific workflow and provenance are a great tool and paradigm not only to better describe, manage, and understand complex scripts with more meaningful structures and explicit data flows, but also to greatly support workflow and data provenance and reproducible computing. Our approach achieves all of the above design objectives, and can apply to both open-source script languages (e.g., R) and closed-source script languages (e.g., MATLAB).

### 3.2. High Level Architecture

The high-level architecture of the integrated system based on our approach is master/slave. VisTrails is the master process and a script language engine such as MATLAB in this work is the slave process, as shown in Figure 3.1.

The basic idea of our system integrating is as follows. First, MATLAB script is wrapped into MATLAB modules which can be added to and used in VisTrails. This is accomplished through an automated MATLAB module generating mechanism that is developed in our integrated VisTrails-MATLAB system. After the wrapped MATLAB modules are generated, they will then be installed into VisTrails to be used in building a scientific workflow in VisTrails as individual workflow modules (i.e., workflow steps). During the course of workflow execution in VisTrails, when starting to execute the first MATLAB module, VisTrails invokes the MATLAB process via MATLAB Engine Application Program Interface (API) from Python to execute the corresponding MATLAB script; the computing output is returned from the MATLAB process to the VisTrails process; VisTrails workflow control selects the next workflow module (workflow step) to execute. This coupling interactions between VisTrails and MATLAB continue until the completion of the entire workflow specified in VisTrails.

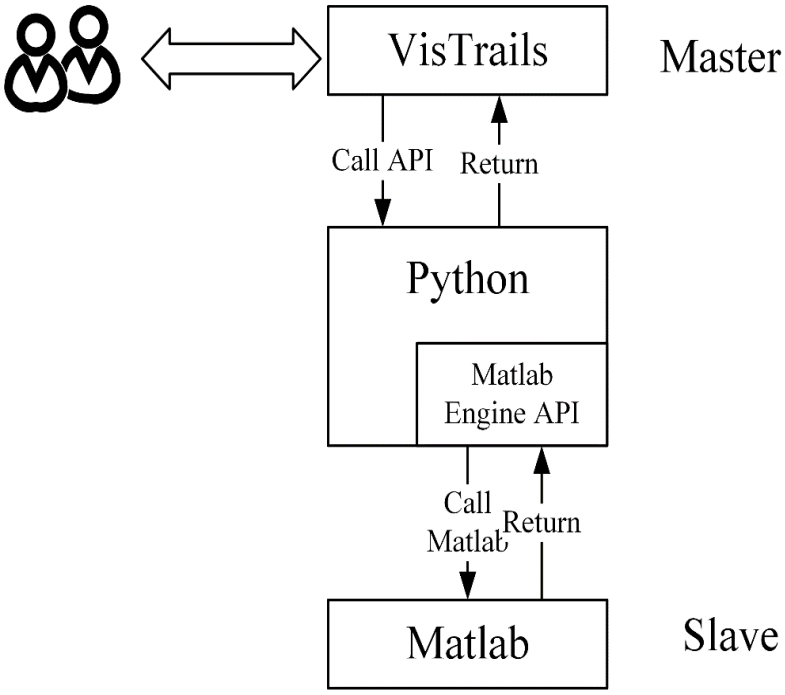


Figure 3.1 Master/Slave Architecture of the Proposed Integrated System.

### 3.3. VisTrails Extension Interface

To integrate VisTrails with MATLAB engine, we need to explore VisTrails extension interface, which is a plugin infrastructure for VisTrails users to integrate user-defined functions and libraries into VisTrails. The added modules are called user packages or user modules. Based on VisTrails extension interface, we provide an automated mechanism to wrap MATLAB script into MATLAB modules and add them into VisTrails for VisTrails-MATLAB integration.

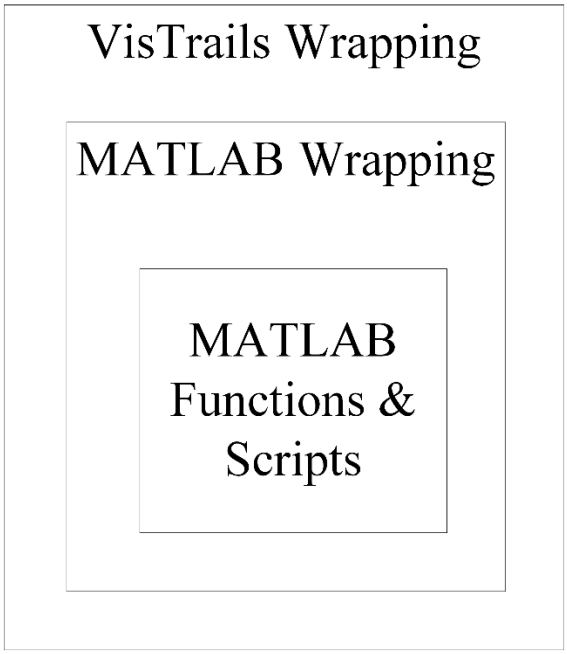


Figure 3.2 Extension structure.

Figure 3.2 is the extension structure. Throughout the integration process, we make two layers wrapping. The outer layer wrapping is VisTrails wrapping. There are two functions to implement this wrapping. The first function is to define a user package. This definition contains module name, input port number, input port type, output port number, output port type, and version and so on. The second function is to implement a compute() method. In this method, it will implement the inner layer wrapping: MATLAB Wrapping. MATLAB Wrapping will use MATLAB Engine API to send data to MATLAB and get data back from MATLAB. Different MATLAB version has different API. This API will be discussed in next section.

The two layers wrapping makes MATLAB function(s) and/or script into a VisTrails user module, which specifies module name, input and output, module version and so on. A new VisTrails user module must be a subclass derived from Module, the VisTrails base class. The compute() method required to be overridden to perform the actual module computation that user specifies by MATLAB functions or script. It uses MATLAB Engine API to send input to MATLAB and get output back from MATLAB, thus integrating MATLAB into VisTrails. For VisTrails wrapping, we need to generate two files, “__init__.py” and “init.py”. These two files make the module reloadable. The

identification, name, version, configuration, and module dependencies methods will be in "__init__.py" file. Imports, other class definitions, input/output port definition, initialization method, and compute method will be in the "init.py" file.

### 3.4. MATLAB Interface

MATLAB has several interfaces for Python. The first one is called MATLAB COM Integration. A COM object is a software component that conforms to the Component Object Model. COM enforces encapsulation of the object, preventing direct access of its data and implementation. COM objects expose interfaces, which consist of properties, methods and events.

Another interface is called DLL (Dynamic Link Library) engine. Users can write their own wrapper on this engine.

The third one is MATLAB Engine API for Python which starting from the MATLAB version 2014b. The MATLAB Engine API for Python provides a package for Python to call MATLAB. It allows MATLAB to be called from other programs in Python, using MATLAB as a computation engine. Engine applications require an installed version of MATLAB.

Sharing MATLAB engine in python is a good feature in MATLAB. It supports the user to connect the MATLAB Engine for Python to a shared MATLAB session that is already running on our local machine. This API allows each individual generated MATLAB module, through its overridden compute( ) method to connect to a shared MATLAB engine for execution.

The second level wrapping in figure 3.2 will use this API to connect to a shared MATLAB engine. The data in VisTrails, MATLAT functions and scripts will be sent to MATLAB via this wrapping. And also, MATLAB returned data will be sent to VisTrails port in this wrapping.

### 3.5. Python Interface

Before MATLAB 2014b, users have to write their own interface to connect to MATLAB. There are several third-party packages in python to support this connection.

### 3.5.1. Pymat

PyMat [13] exposes the MATLAB engine interface allowing Python programs to start, close, and communicate with a MATLAB engine session. In addition, the package allows transferring matrices to and from an MATLAB workspace. These matrices can be specified as NumPy arrays, allowing a blend between the mathematical capabilities of NumPy and those of MATLAB. The drawback of Pymat is that it does not support calling MATLAB function.

It is a low-level interface to MATLAB using the MATLAB engine (libeng) for communication. The module has to be compiled and linked with libeng.

### 3.5.2. Mlabwrap

Mlabwrap is a high-level python to MATLAB bridge that lets Matlab look like a normal python library [14]. It is also coming as a module which needs compilation and linking against libeng. It exposes Matlab functions to python so user can call MATLAB functions.

### 3.5.3. Pymatlab

This package lets Python users interface and communicate with MATLAB from Python [15]. Pymatlab makes it easier for users to integrate a project with a large MATLAB codebase into python scripts by using MATLAB scripts as a part of the python program.

The basic functionality of this package is to send data from Python to MATLAB's workspace to be able to run MATLAB function on the data. After processing, user retrieve back data to python. This enables user to process data with MATLAB's built in functions, toolboxes or MATLAB-scripts. It is also possible to use MATLAB's to generate plots or other graphics.

The package uses Numpy's ndarrays and translates them into MATLAB's mxarrays using Python's ctypes and MATLAB's mx library. The interface to MATLAB's workspace in done through MATLAB's engine library.

This package also interacts with Matlab through libeng. Unlike the other packages this one loads the engine library through ctypes thus no compilation required. In our research, we use this third-party package as our interface for old MATLAB version. (MATLAB 7.0)

# CHAPTER 4. IMPLEMENTATION

As described in Chapter 3, our solution to VisTrails-MATLAB integration is a two-phase procedure. The first phase is how to build MATLAB modules from MATLAB script, and install these MATLAB modules into VisTrails as basic building blocks for establishing a scientific workflow. The second phase is the execution of a workflow by the integrated VisTrails-MATLAB system. Figure 4.1 illustrates our VisTrails-MATLAB integration process in some detail. Step 1 indicates wrapping MATLAB script and generating MATLAB modules, while step 2 indicates installing MATLAB modules into VisTrails. Step 3 indicates building a scientific workflow in VisTrails with MATLAB modules, just like VisTrails self-generated modules. These three steps form the first phase of the integration procedure. Steps 4-7 illustrate the course of execution of the built scientific workflow with MATLAB modules, the second phase of the integration procedure, in which VisTrails will automatically record each workflow step with its parameter setting, data input and data output during the execution as provenance information. The key development in our solution is an automated MATLAB module generating mechanism which can accomplish step 1 in Figure 4.1, while the rest steps shown in Figure 4.1 are handled the by VisTrails' own functionality in conjunction with MATLAB Engine API for Python.

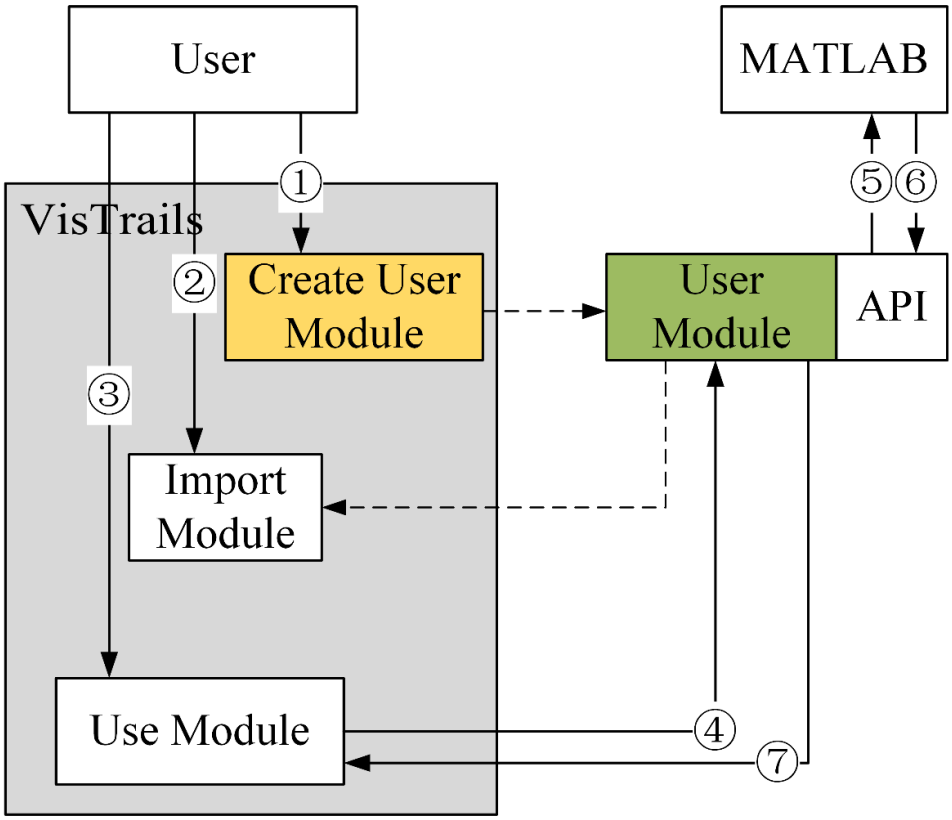


Figure 4.1 Illustration of VisTrails-MATLAB integration.

In the second step, the user, through the VisTrails menu in the preference option, installs any MATLAB module that is generated in step 1. Installed MATLAB modules can be used in VisTrails in the form of workflow, enabling MATLAB script data provenance via VisTrails provenance functionality at the workflow module level. In the third step, MATLAB modules are used like VisTrails self-generated module, free to drag and drop any connection of data flow between modules via VisTrails Graphical User Interface, generating a scientific workflow. When performing the MATLAB module in steps 4 and 5, VisTrails calls the MATLAB API in python and passes the corresponding input data and commands to the MATLAB engine for execution, and the output data are returned to VisTrails in steps 6 and 7. During the execution, VisTrails will automatically record all the operations and data provenance at run time.

## 4.1. Generation of MATLAB Module

As VisTrails is the master part of the integrated VisTrails-MATLAB, our solution is implemented based on VisTrails (version 2.1.4). We have developed multiple VisTrails-MATLAB integration versions to support different MATLAB versions from MATLAB 7 to MATLAB 2016b. As the new version of MATLAB is the mainstream, we first present our implementation with MATLAB 2016b.

### 4.1.1. With MATLAB 2016b

First of all, we extend VisTrails' Graphical User Interface (GUI) to include a MATLAB menu for generating MATLAB modules in VisTrails. The graphical interface of VisTrails is built by Tkinter, the de-facto standard Python GUI interface, which is a thin object-oriented layer on top of Tcl/Tk. Both Tkinter and Tcl/Tk are available on most UNIX platforms, as well as on Windows systems. By modifying the VisTrails GUI file, we add a menu button for MATLAB module generation into the VisTrails user interface's menu bar, as shown in Figure 4.2.

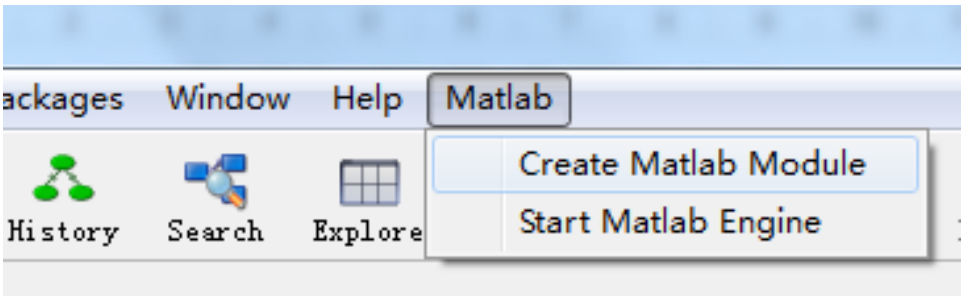


Figure 4.2 Illustration of integrated VisTrails-MATLAB menu bar.

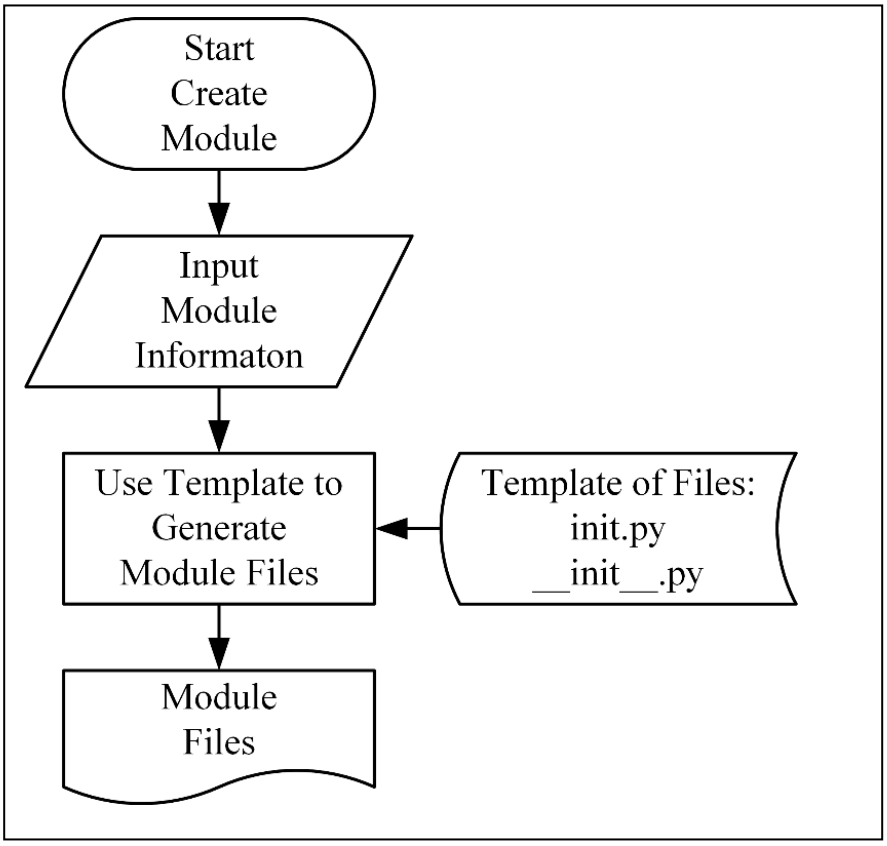


Figure 4.3 The process of generating MATLAB module.

The process of generating MATLAB module is described in Figure 4.3. First, from the VisTrails-MATLAB menu bar, click Matlab button and select “Create Matlab Module”. The user then enters necessary information including MATLAB module name, input and output ports and port types. Here the module name can be a MATLAB function name, a user's own defined function or MATLAB script M file name. The input and output ports and types correspond to the number and types of inputs and outputs of the corresponding module. Finally, after the completion of the above information, to the user starts MATLAB module generating. At this moment, generator.py file is called and executed, which will process the inputted information of MATLAB module, use the template file, and then generate the corresponding module files. Two files will be generated for the user’s MATLAB module: init.py and __init__.py. Since each generated file follows some common patterns, we use string template to provide simple string substitutions. Thus, two template files are used for the generated module: “init.template” file, for generating the init.py file of the module; and “ __init__.template” file, for

generating the file "__ init__.py", which containing the ID and version information. Finally, these two generated files will be saved in the user package directory: %USERPROFILE%\.vistrails\userpackages\. There can be different subfolders with different names under this folder, corresponding to user's different generated MATLAB modules.

### 4.1.2. With MATLAB 7

As we mentioned in 3.5. There is no official interface for Python to MATLAB. So we use a third party open source package - Pymatlab as our python interface.

The main steps in generating MATLAB are the same with MATLAB 2016b. The only differences are the "template" files, which need to be changed to another version.

## 4.2. Template Design

### 4.2.1. With MATLAB 2016b

Two templates designed to help create MATLAB modules in VisTrails are shown in Figures 4.4 and 4.5, respectively.

```
1  identifier = 'My.${ModuleName}'
2  name = '${ModuleName}'
3  version = '0.0.1'
```

Figure 4.4 __init__.template file.

Template file "__init__.template" is designed and used to generate __init__.py file, which provides metadata about a module. ${ … } is a substitution placeholder. Name is a human-readable name for the module. Version is simply the version information about the module. The most important part of the metadata here is the identifier, stored in the identifier variable. This string variable must be globally unique across all modules in VisTrails SWfMS, not only in MATLAB modules.

```
1  import vistrails.core.modules.module_registry
2  from vistrails.core.modules.vistrails_module import Module, ModuleError
3  from vistrails.core.modules.module_registry import get_module_registry
4  import matlab.engine
5  ##########################################################################
6  class ${ModuleName}(Module):
7      def __init__(self):
9
10     def compute(self):
11         eng = matlab.engine.connect_matlab()
12
13         ${GetInputData}
14
15         ${PutValue}
16
17         ${Run}
18
19         ${GetValue}
20
21         ${SetOutputData}
22 ##########################################################################
23 def initialize(*args, **keywords):
31
```

Figure 4.5 init.template.file.

Template file “init.template” is designed and used to generate init.py file. The actual definitions information of the modules is contained in the init.py file. Each VisTrails module corresponds to a Python class which derives from the Module class. The Module class is defined in vistrails.core.modules.vistrails_module. Each VisTrails user module must define its input ports and output ports, and also have to implement a compute() method that takes no extra parameters. In compute() method, we defined several placeholders. We developed a “Generate module”, which will generate strings and provide string substitution to all these placeholders.

${GetInputDate} is the placeholder for setting the input port data to this module. ${PutValue} is the placeholder for putting input values to MATLAB workspace. ${Run} is the placeholder for running MATLAB engine. ${GetValue} is the placeholder for getting value form MATLAB engine and saving the value to local variable. ${SetOutputData} is the placeholder for sending data to output port of this module.

Function initialize() is used to register input ports and output ports. In this function, it contains placeholder ${RegInputPort} and ${RegOutputPort}, which will also be substituted during module generating.

### 4.2.2. With MATLAB 7

```
import vistrails.core.modules.module_registry
from vistrails.core.modules.vistrails_module import Module, ModuleError
from vistrails.core.modules.module_registry import get_module_registry
import pymatlab as py
eng=py.session_factory()

##############################################################################

# ${ModuleName}

class ${ModuleName}(Module):
    def __init__(self):
        Module.__init__(self)

    def compute(self):
        ${GetInputData}

        ${PutValue}

        ${RunEval}

        ${Run}

        ${GetValue}

        ${SetOutputData}

##############################################################################
def initialize(*args, **keywords):
    reg = get_module_registry()

    reg.add_module(${ModuleName})

    ${RegInputPort}

    ${RegOutputPort}
```

Figure 4.6 init.template file for MATLAB 7.

For MATLAB 7, the “init.template” file is changed to another version. Shown in Figure 4.6. “__init__.template” file is the same with MATLAB 2016b.

### 4.3. Graphical User Interface Design

We have designed two different user interfaces for inputting MATLAB module information. One is the command line input, whereas the other one is the GUI input. For user's convenience, we integrate GUI method into VisTrails. In the menu bar click MATLAB-> Create Matlab Module, the GUI interface will be displayed, as shown in Figure 4.6.

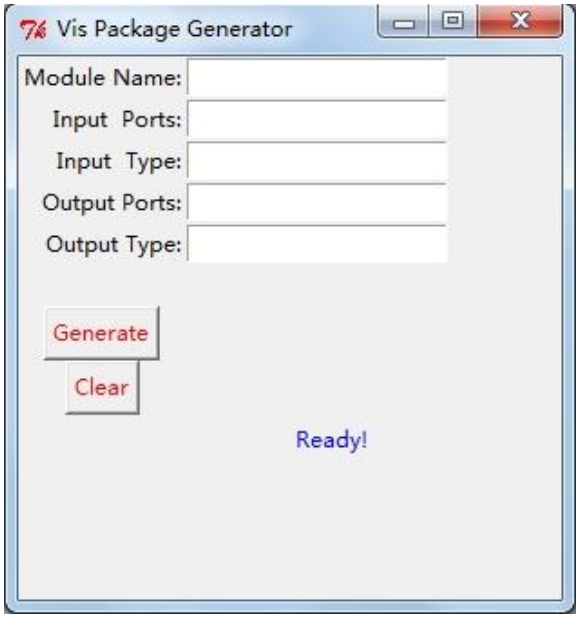


Figure 4.7 The Graphical User Interface for inputting MATLAB module information.

The Module Name can be a specific function name in MATLAB, such as wavedec2, or the user defined script m file. For example, if the user has a script file called wavelet_analysis.m, the Module Name is wavelet_analysis.

Input Ports and Output Ports indicate the number of parameters and data input and output of the module. These input and output ports can be used to connect different modules in a workflow via mouse dragging. If a module does not have any parameter/data to input or output, one can enter 0 directly in the corresponding GUI entry.

Input Type and Output Type correspond to the parameter types of the module input and output, respectively. These types are of basic types in VisTrails, and Table 4.1

lists the type mapping from VisTrails types to MATLAB types. Table 4.2 list the type mapping from MATLAB types to VisTrails types.

If a type used in MATLAB has no counterpart type in VisTrails, we then create a new type name to support it. For example, a new “Func” type in VisTrails is created for function handle type in MATLAB. When MATLAB functions return output arguments, the MATLAB Engine API for Python converts the data types into equivalent Python data types.

Table 4.1: Data type mapping from VisTrails to MATLAB

| **VisTrails data type** | **MATLAB data type** |
|---|---|
| Float | Double |
| Integer | Int64 |
| List | Numerical Array |
| String | Char array |
| Boolean | Logical |
| Func | Function handle |
| Dictionary | Structure |

Table 4.2: Data type mapping from MATLAB to VisTrails

| MATLAB Output Argument Type | Resulting VisTrails Data Type |
|---|---|
| Double | Float |
| Single | Float |
| Int8 | Integer |
| Uint8 | Integer |
| Int16 | Integer |
| Uint16 | Integer |
| Int32 | Integer |
| Int64 | Integer |
| Logical | Boolean |
| Char array | String |
| Structure | Dict |
| Numerical array | List |

Finally, the Generate button is to start generating the two files for MATLAB module, while the Clear button is to clear all the input content in the above entries.

In this section with MATLAB 7, it has the same interface design.

### 4.4. Other Extensions

For the convenience of users, we have also developed several generic MATLAB modules listed in Table 4.3. These modules are pre-installed in our integrated VisTrails-MATLAB system and are ready for use.

Table 4.3: Generic modules

| Module Name | Features |
|---|---|
| Load | Load is a file operation module. Read MATLAB .mat file's data to VisTrails Workflow |
| Save | Save is a file operation module. Save VisTrails Workflow's data to MATLAB .mat file. |
| MatlabSource | MatlabSource is a module that executes an arbitrary piece of MATLAB code. It is especially useful for one-off pieces of 'glue' in a pipeline. |
| MatlabBinaryOp | MatlabBinaryOp is a module that performs binary operations on its inputs. These operations contains matrix operations in MATLAB. |
| Mkdir | Mkdir is a make directory command module. The only input is the name of directory. |

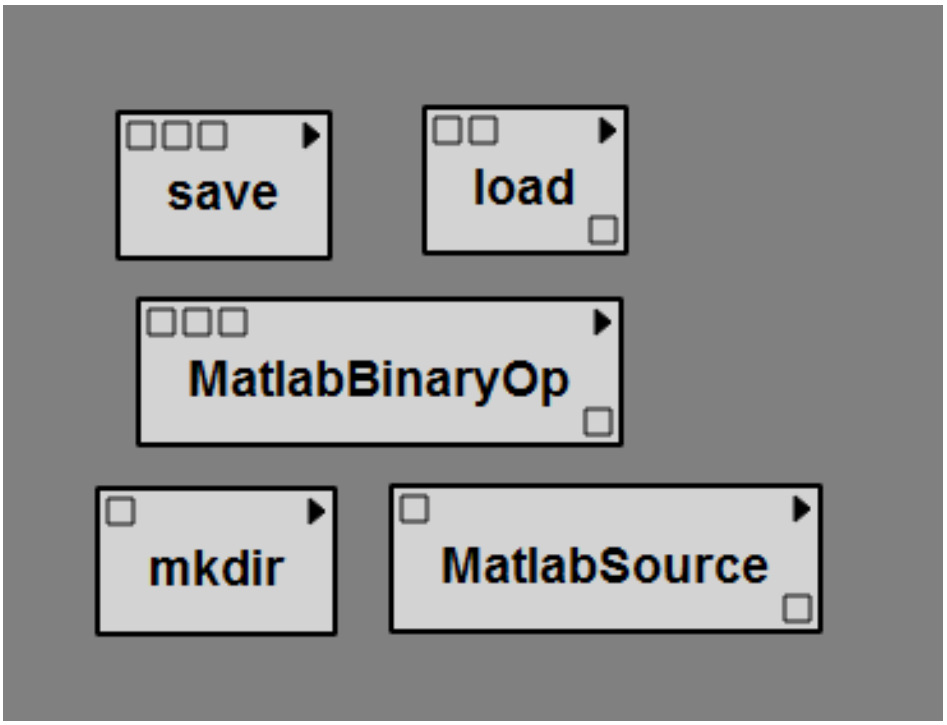


Figure 4.8 Some generic MATLAB modules developed for convenient use.

In Figure 4.8 there are several generic modules.

“Save” is a file operation module. Save VisTrails Workflow’s data to MATLAB .mat file. The first input port is for the data that will be saved. The second input port is for .mat file name. The third input port is for variable name in .mat file.

“Load” is a file operation module. Read MATLAB .mat file’s data to VisTrails Workflow. The first input is for .mat file name. The second input is for variable name. The output port is to output data that it just read.

“MatlabBinaryOp” is a module that performs binary operations on its inputs. These operations contain matrix operations in MATLAB. The first input is for the operation symbol. The second and third input are the two values. The only output is the result.

“Mkdir” is a make directory command module. The only input is the name of directory.

“MatlabSource” is a module that executes an arbitrary piece of MATLAB code. It is especially useful for one-off pieces of 'glue' in a pipeline. The input is the command line, which type is string. The output is the result that return from MATLAB engine.

“Mkdir” and “Save”, these two modules require system administrator privileges

# CHAPTER 5. DEMONSTRATION

## 5.1. Demonstration Environments

In this chapter, we will demonstrate our VisTrails-MATLAB system in details. Before showing the demonstration, we should first introduce the experiment software environment.

Table 5.1: Experiment environment

| Software | Version |
|---|---|
| MATLAB | R2016b (9.1.0.441655)<br>64-bit (win64) |
| VisTrails | 2.1.4.269e4808eca3 |
| Operation System | Windows 7 Ultimate x64 |
| Python | 2.7.3 (64bit) |

## 5.2. Main Functions overview

### 5.2.1. From MATLAB Script to VisTrails' Workflow

Demonstration scenario 1: User has a complete script in MATLAB, which requires scientific workflow to achieve data provenance and version management.

To demonstrate, we use a MATLAB program for image compression based on wavelet transformation. In order to record the data in the program, and to test the impact of different wavelet families on the compression results, we need to record different parameters and results.

```
1 –   X=imread('lena512.bmp');
2 –   [c,s]=wavedec2(X,2,'bior3.7');
3
4 –   ca1=appcoef2(c,s,'bior3.7',1);
5 –   ca1=wcodemat(ca1,440,'mat',0);
6 –   subplot(121);imshow(ca1,[]);
7 –   title('First Compression');
8 –   imwrite(ca1,'1.jpg');
9
10 –  ca2=appcoef2(c,s,'bior3.7',2);
11 –  ca2=wcodemat(ca2,440,'mat',0);
12 –  subplot(122);imshow(ca2,[]);
13 –  title('Second Compression');
14 –  imwrite(ca2,'2.jpg');
```

I

II

Figure 5.1 A MATLAB script for image compression based on wavelet transformation

In MATLAB, there are typically two ways to record different parameters and their corresponding results. One is to copy scripts multiple times and use different parameters in each script, and store the results in different variables, finally output all the results and compare them. In Figure 5.1, segment I computes the approximation coefficients at level 1 and saves the result in ca1, while segment II computes the approximation coefficients at level 2 and saves the result in ca2. The other is to run section I multiple times by using different parameters, and save the corresponding results in different files.

Both above methods have management flaws. They all need to manually maintain the correspondence relationship (e.g., through file naming) between the input parameters and the experimental results for multiple trials, which is error prone and do not scale. Once the manual recording becomes erroneous, one simply cannot reproduce the experimental process and results.

Our integrated VisTrails-MATLAB system with its provenance can be a desirable solution to the above problem. One can wrap the above MATLAB code into MATLAB modules and execute them in the environment of integrated VisTrails-MATLAB system. Whenever the user changes parameters of any module, VisTrails will generate a new

corresponding sub-version. Each version automatically records all parameter configurations and their correspondence to the obtained results at the run time.

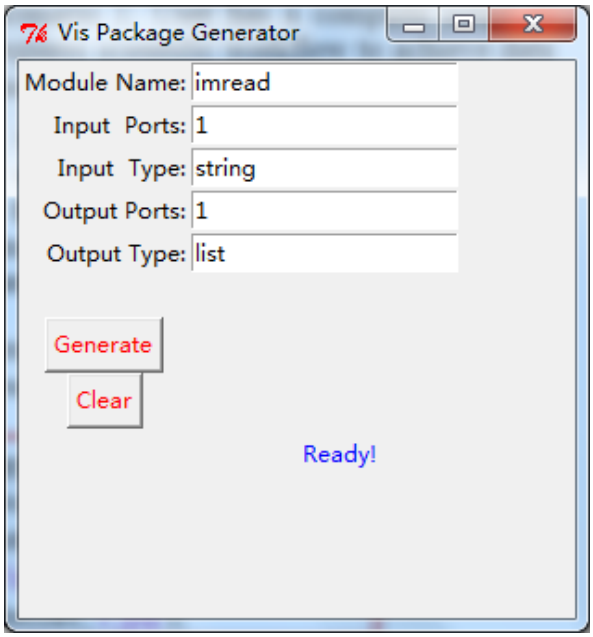


Figure 5.2 Creating module in VisTrails

Figure 5.2 shows how to create “imread” MATLAB module from MATLAB script in the integrated VisTrails-MATLAB. All the other MATLAB modules, such as “wavedec2”, “appcoef2”, “wcodemat”, “imwrite”, can be created in a similar way. After enable, these module-packages in VisTrails (see Figure 5.3) and connect the corresponding ports, we get the workflow specified in VisTrails shown in Figure 5.4.

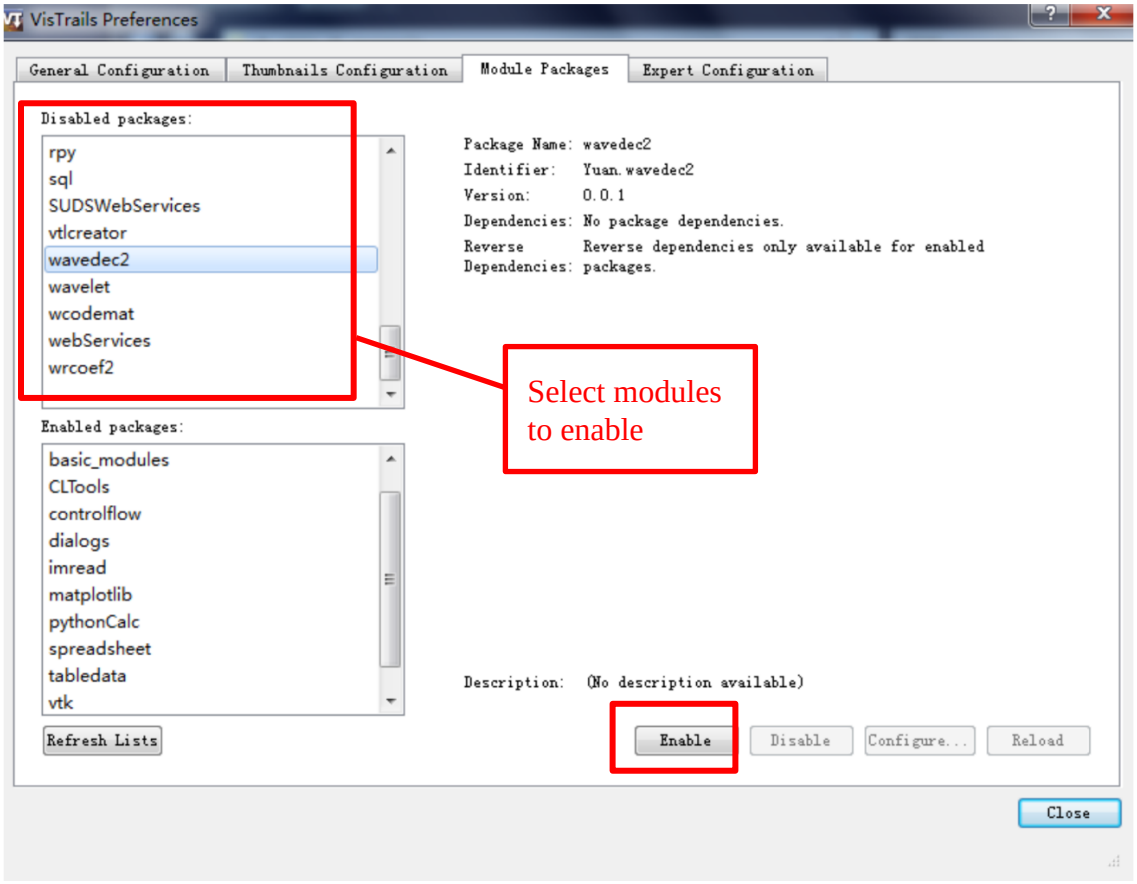


Figure 5.3 Enabling module packages in VisTrails

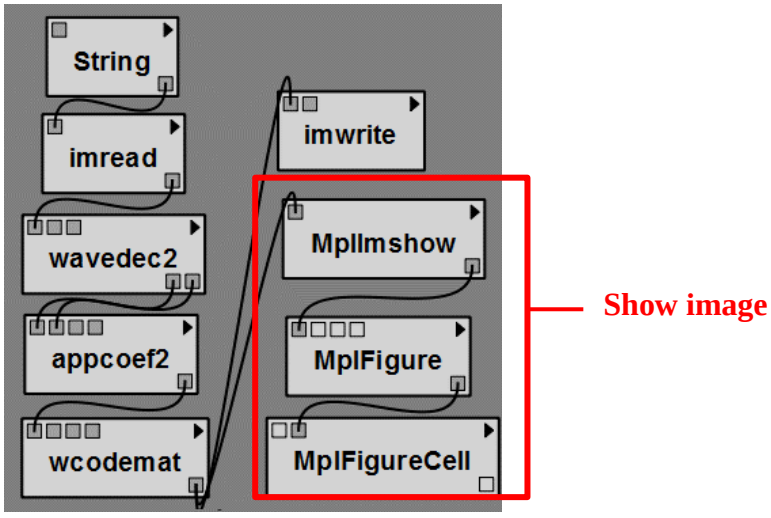


Figure 5.4 Illustration of built workflow in integrated VisTrails-MATLAB system

After forming the workflow, one can set parameters. As shown in Figure 5.5, one can set module parameters in “wavedec2” when this module is selected. The first input port is the image data that just read from “imread”. The second input port is the level of decomposition. Here we use level 2. The third input is the wavelet named in string. Here we use “bior3.7”.

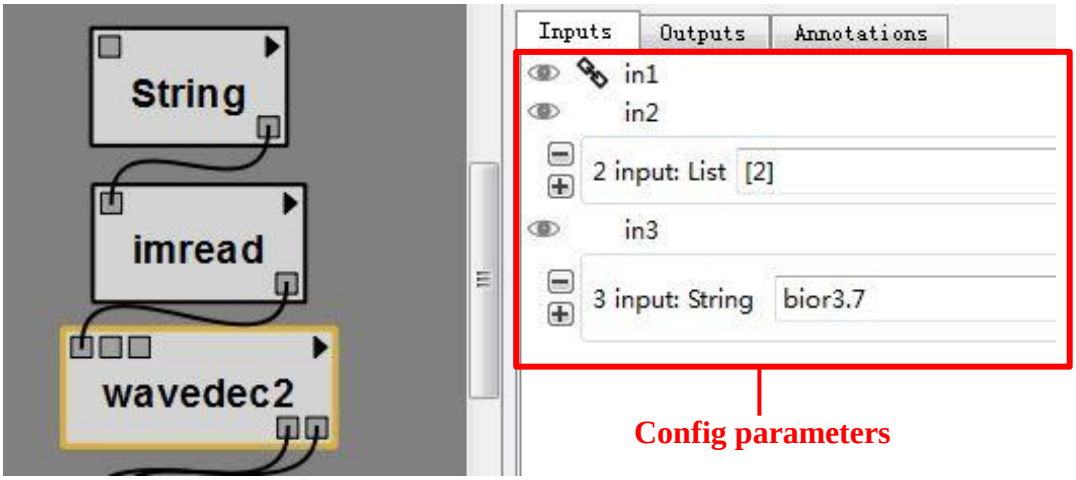


Figure 5.5 Set parameters in module wavedec2

When one wants to modify the parameters, VisTrails will record our trail and parameters. It will be shown in history through a tree structure. One can rename each version to facilitate our management.

The following version tree shown in Figure5.6 is generated based on the two execution trials of the workflow given in Figure 5.4, corresponding to the original MATLAB script in Figure 5.1. The “first_compression” computes the approximation coefficients at level 1, and the “second_compression” computes the approximation coefficients at level 2.

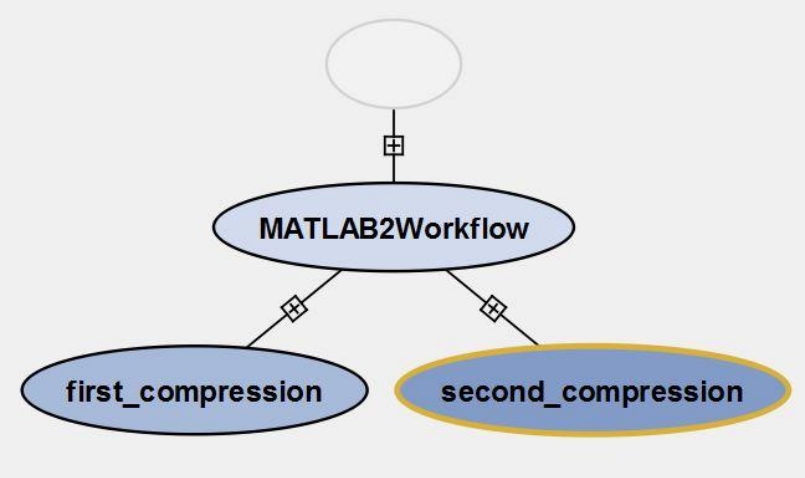


Figure 5.6 Illustration of version tree in integrated VisTrails-MATLAB system.

The integrated VisTrails-MATLAB can significantly improve the runtime efficiency of computational trials due to its data provenance. Modules whose parameters and input are not modified do not need to be re-run in a new trail, the workflow can directly use the intermediate data result that has been cached. The workflow can be very convenient to achieve module reuse. The user does not need to copy and paste the code, but only needs to modify the parameters and/or reconnect modules via their input and output port(s) for each new trail. After that, a new version of workflow will be created. Moreover, for different workflow versions with different parameter settings, VisTrails automatically records all the changes and the corresponding results. The user can easily query between different versions and can also add their own tags and notes for different versions, facilitating reproducible computing and research.

The user's computational experiments generate a number of versions, each of which corresponds to different parameters and corresponding results. How can the user quickly find the corresponding parameter settings through the output results, or find the corresponding output result file through the parameter settings via VisTrails version tree provenance?

First, in each workflow version, VisTrails automatically records all the changes, parameters and time. When the user wants to check provenance information, he/she can browse the version tree, and double click on the chosen workflow version node, say first_compression, of the version tree (see Figure 5.7 and Figure 5.8). If he/she wants to

check the results from this workflow, he/she can select module imwrite to get the image file directory information as shown on right side in Figure 5.9. If he/she wants to check parameter setting for this result, he/she selects the module which contains the parameters of interest.

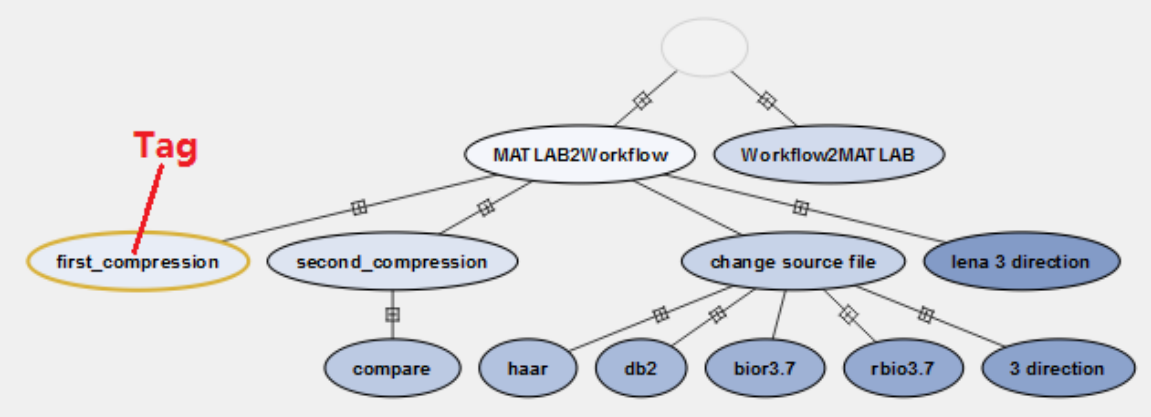


Figure 5.7 Version tree management left side.

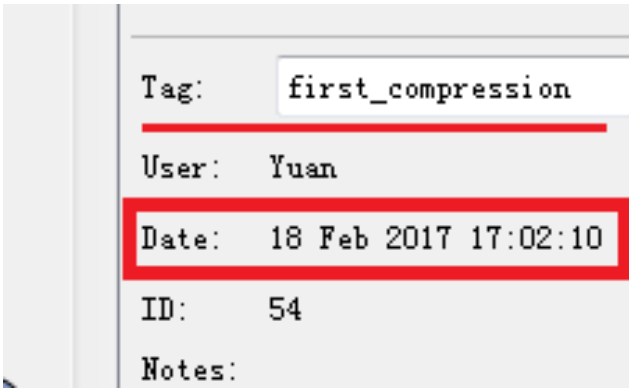


Figure 5.8 Version tree management right side.

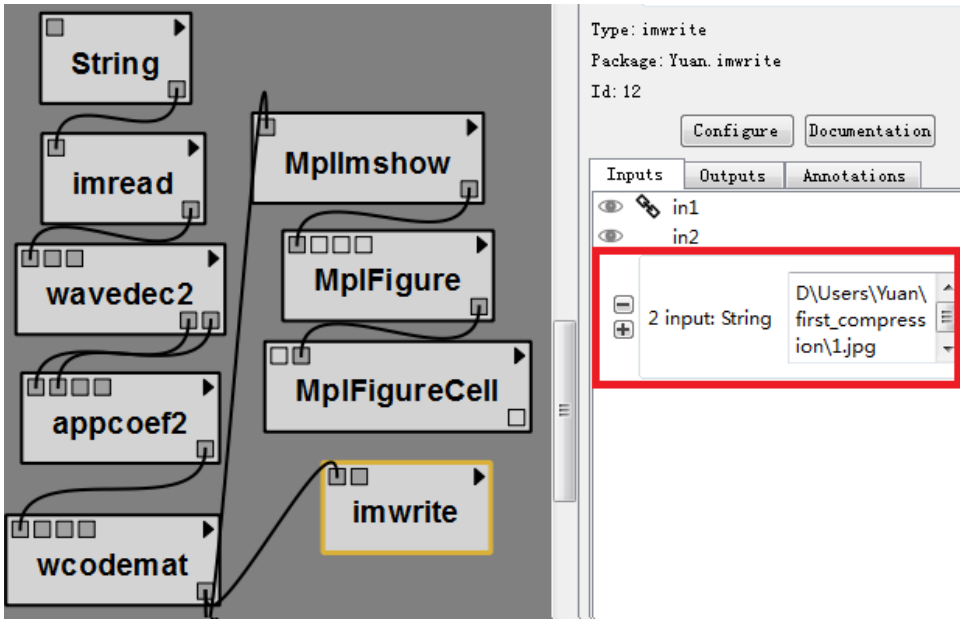


Figure 5.9 Finding the result saving directory.

A tag can be given in the Tag field for user's convenience. The panel displays information about the user who created the selected workflow version and when that version was created (see Figure 5.8). In addition, the Notes field allows users to write notes or annotations associated with a version. Notes are automatically saved when one saves the VisTrails file.

### 5.2.2. From VisTrails' Workflow to MATLAB Script

Demonstration scenario 2: One can use workflow to design a computational experiment using MATLAB script, to achieve better computational structure and dataflow at the workflow level. For the chunk codes that MATLAB already existed, we can create MATLAB modules to wrap these codes, thus simplifying the design process.

Still taking the wavelet compression as an example, the workflow of image compression study can include the following workflow steps: image reading, frequency decomposition, wavelet transformation, image generation, image output, file output these six parts. The first three parts are the scripts from MATLAB, which can be generated very conveniently from MATLAB script. For these parts that logically together we can create a module in the exploratory study, this MATLAB module is at a coarser level of

granularity level. For example, we can use a user-defined m file, which contains a large chunk of script, to be used as a workflow step.

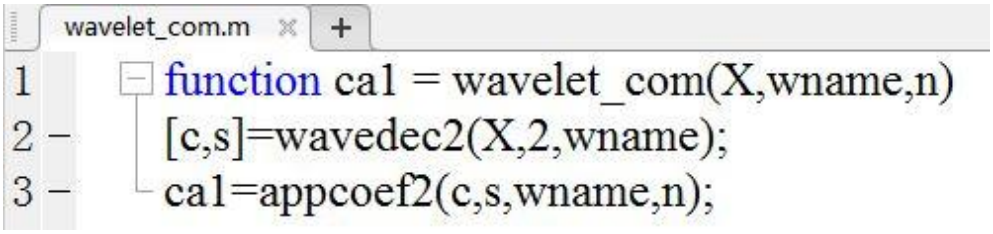


Figure 5.10 User function in MATLAB.

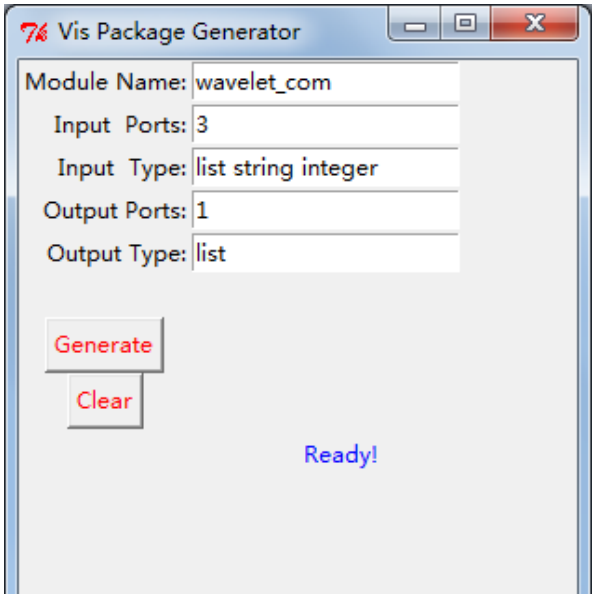


Figure 5.11 Use user function as workflow module

Figure 5.10 shows a user function in MATLAB, where user can create a MATLAB function to wrap these in an m file. After that, the user can create a corresponding MATLAB module in VisTrails, as shown in Figure 5.11. There are 3 input variables. The first one is image data which is list type. The second one is wavelet name which is a string type. The third one is wavelet level which is an integer type. The output is an image data list after wavelet compression.

The module wavelet_com shown in figure 5.12, is the module we created from the MATLAB user function. We can configure the parameters on the right side. Port in2 is the wavelet name, port in3 is the wavelet level. Under the wavelet_com module is the Show results module which contains image generation, image output and image save modules.

This coarser level of granularity module abstract method can facilitate us to design experiment. This level granularity will also reduce the overhead of provenance recording.

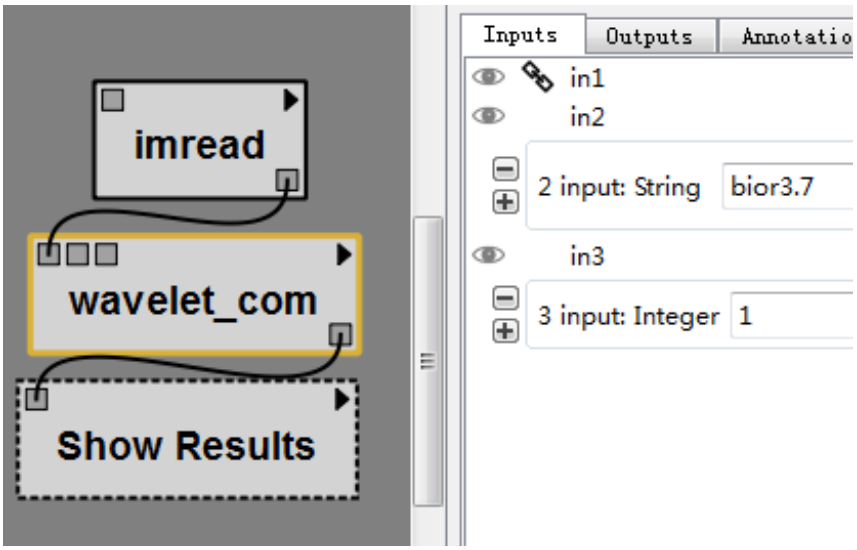


Figure 5.12 Use of user function as a workflow module

### 5.2.3. Two way search

The user's computational experiment generates a number of versions, each of which corresponds to different parameters and different results. How can the user quickly find the corresponding parameter settings through the output results, or find the corresponding output result file through the parameter settings?

The traditional method is: after changing the parameters, the user saves the corresponding result in a separate folder and manually write a readme file to record the changes. Compared to the traditional method, VisTrails use version tree provenance to

manage each change. Notes and tags can be written in the version tree so that to facilitate the search and management.

First, in each version, write the tag area with parameters that generate this version. So that users can quickly get the parameters of each version when browsing the version tree.

Second, write the note as the name of the subfolder to facilitate location of the file. When a user looks for parameters that produce this result, the parameter is in tag of this version.

If users want to find the result via parameters, they can browse the version tree. Each node in the version tree contains a brief describe we added in note, and tag area contains detains detail parameters. So we can use note to find the folder that the result saved. Also, VisTrails has preview window, which can show the result image.

### 5.3. Wavelet Image Compression Example

Image compression refers to the use of as few data as possible to indicate the source of the image signal. There are a lot of compression methods, which can be divided into lossless compression and lossy compression. The former technology can accurately reconstruct the image, the compression is relatively low. The lossy compression technology will introduce distortion, but it is not obvious distortion.

The wavelet transform can be used to decompose the image. The wavelet decomposition is complete and orthogonal. The wavelet transform gives a time-frequency window that can be adjusted so that it is more accurate to the time at high frequencies, capturing the details and edges of the image; positioning the frequency at low frequencies can reflect the overall image feature.

Wavelet transform based image compression technology using multi-level scale analysis. According to their important of different levels of different coefficients, it is easily to get a high compression ratio. After an image is decomposed by two-dimensional discrete wavelet transform, we obtain get a series of sub-images with different resolutions. The frequency of sub-images with different resolutions is different. And the high resolution (i.e., high frequency) sub-image on most of the points are close to 0. The higher the frequency the more obvious this phenomenon.

Now, we can use VisTrails-MATLAB system to implement the wavelet image compression experiment.

Firstly, use Create Matlab Module GUI to create all the MATLAB toolboxes and function that we need (Figure 4.2). Then enable all the user modules that we just created (Figure 5.13). After that, connect all the modules and configure the parameters in each module (Figure 5.14) and execute the workflow.

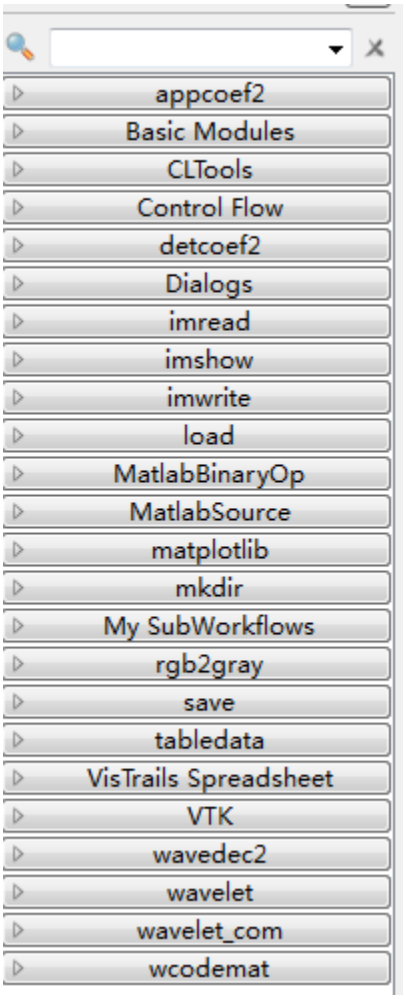


Figure 5.13 User Modules.

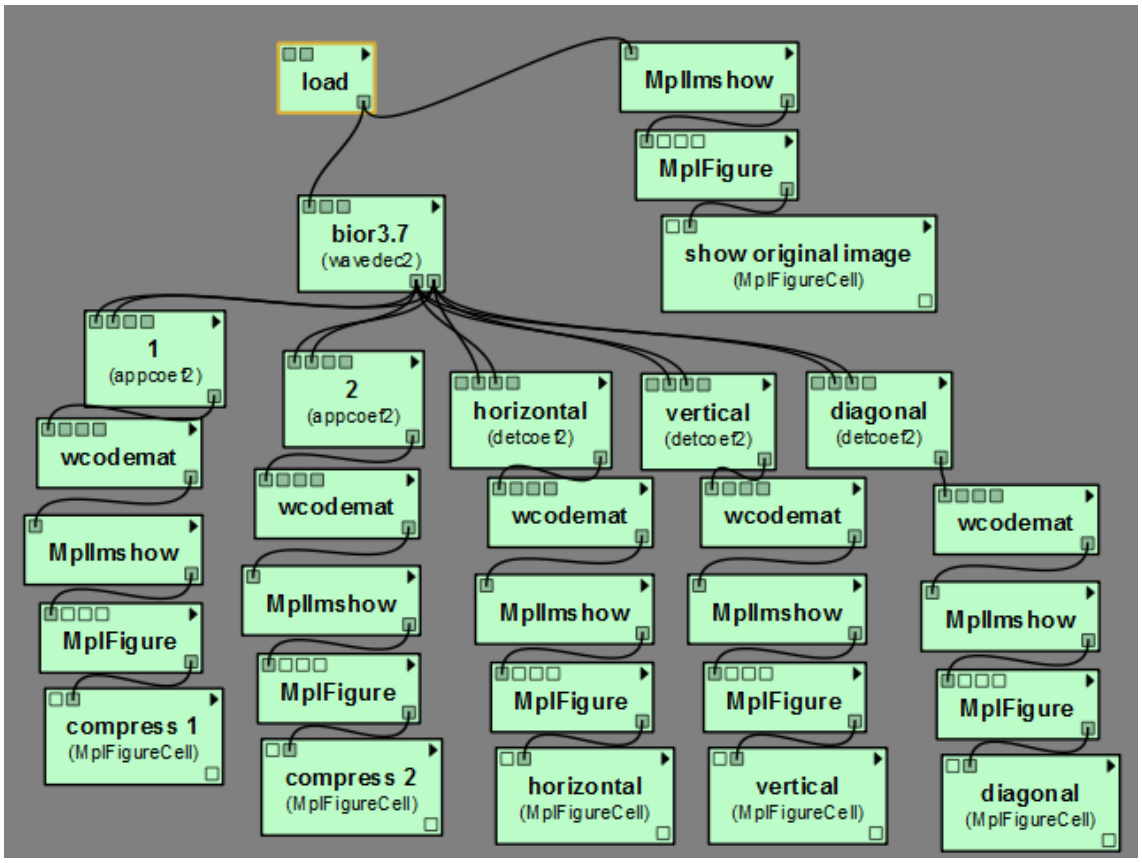


Figure 5.14 Connect all modules.

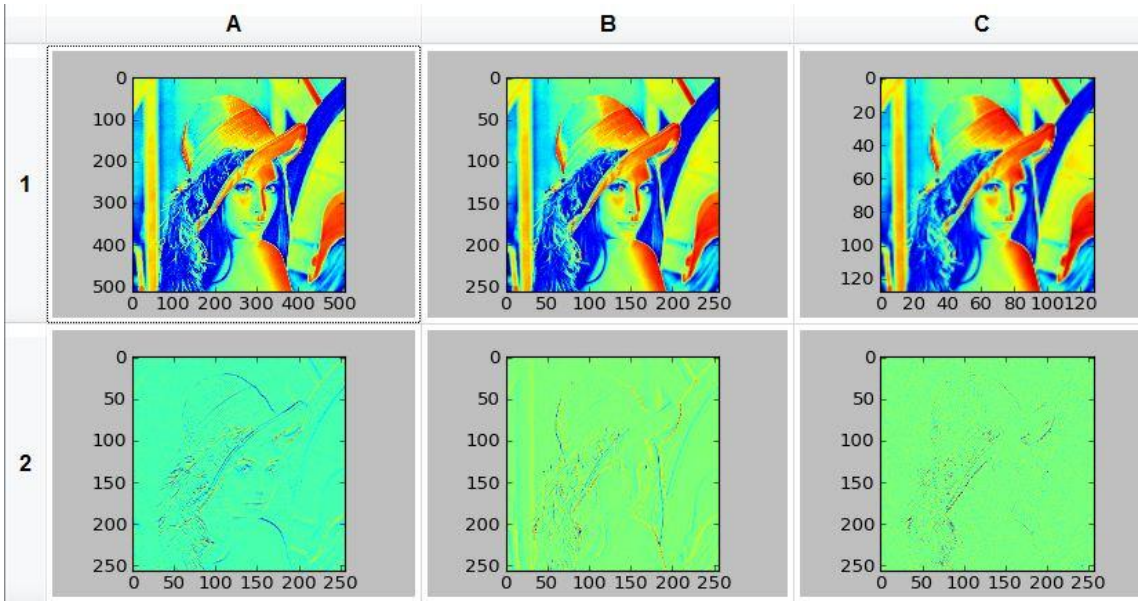


Figure 5.15 Compression results.

Figure 5.15 shows the compression results that we use “bior3.7” as wavelet filter. 1A is the original image. 1B is the result after extract the first level of low-frequency information. 1C is the result after extract the second level of low-frequency information. 2A~2C extract from the wavelet decomposition structure the horizontal, vertical, or diagonal detail coefficient.

The original image 1A size is 263,222 bytes. The first level compression image 1B size is 70,510 bytes, compression ratio is 73.21%. The second level compression image 1C size is 20,538 bytes, compression ratio is 92.20%

All the other wavelet filters are shown in table 5.2. We can use different wavelet filter to do exploratory research. All the results will be saved in the version trees, shown in Figure 5.16.

Table 5.2: Wavelet filter

| Wavelet Families | Wavelet Families |
|---|---|
| Daubechies | 'db1' or 'haar', 'db2', ... ,'db10', ... , 'db45' |
| Coiflets | 'coif1', ... , 'coif5' |
| Symlets | 'sym2', ... , 'sym8', ... ,'sym45' |
| Fejer-Korovkin filters | 'fk4', 'fk6', 'fk8', 'fk14', 'fk22' |
| Discrete Meyer | 'dmey' |
| Biorthogonal | 'bior1.1', 'bior1.3', 'bior1.5'<br>'bior2.2', 'bior2.4', 'bior2.6', 'bior2.8'<br>'bior3.1', 'bior3.3', 'bior3.5', 'bior3.7'<br>'bior3.9', 'bior4.4', 'bior5.5', 'bior6.8' |
| Reverse Biorthogonal | 'rbio1.1', 'rbio1.3', 'rbio1.5'<br>'rbio2.2', 'rbio2.4', 'rbio2.6', 'rbio2.8'<br>'rbio3.1', 'rbio3.3', 'rbio3.5', 'rbio3.7'<br>'rbio3.9', 'rbio4.4', 'rbio5.5', 'rbio6.8' |

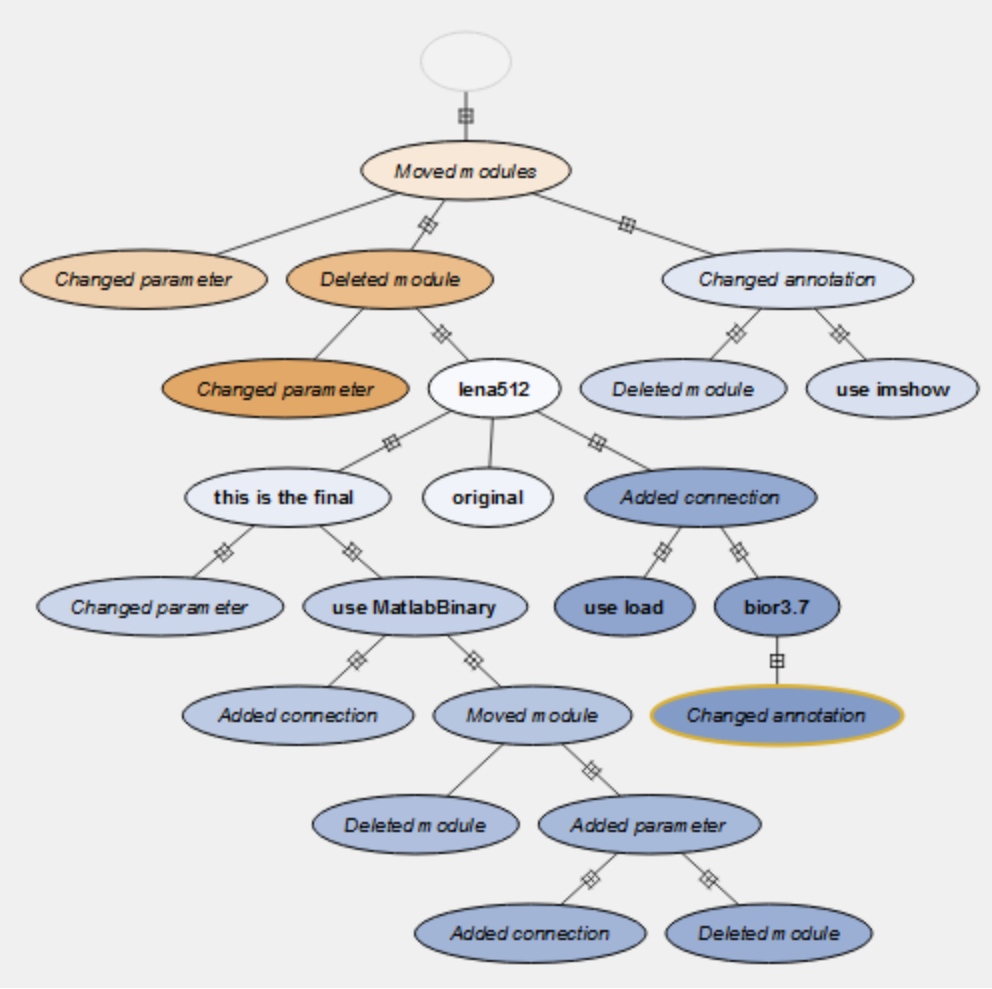


Figure 5.16 Version tree.

# CHAPTER 6. CONCLUSION AND FUTURE WORK

In this work, we presented a general approach to provenance and scientific workflows for script languages, to facilitate reproducible computing and research. We demonstrated our approach by integrating MATLAB with an open source scientific workflow system VisTrails, to support scientific workflows and provenance for MATLAB scripts. We described the design and implementation of our integrated system solution in detail. Our approach extends the advantages of VisTrails to widely used script languages and promotes reproducibility in exploratory research for scientists and engineers to address the challenges of the verification, validation, evaluation and sharing of any new scientific result in broad community.

One direction we plan to explore in future work is how to evaluate the cost of the effectiveness of our proposal, in particular since in some cases it may require communication from scripting language engine with an existing scientific workflow system. The time consumption will be very different in different situations. In some cases, the first time running the SWfMS system will cost more resources and time in communication. While in other cases, when user modify some parameters in workflow, the rest of workflow data had already been cached. It saves more time resources.

We also plan to integrate other scripting languages with VisTrails and other scientific workflow systems. So that users of different habits can enjoy the benefits of SWfMS.

# REFERENCES


[1] V. Stodden, F. Leisch, and R. D. Peng, Implementing reproducible research. Chapman and Hall CRC, 2014.

[2] P. Missier, S. Woodman, H. Hiden, and P. Watson, “Provenance and data differencing for workflow reproducibility analysis,” *Concurrency and Computation: Practice and Experience*, 2013.

[3] J. Freire, D. Koop, F. Chirigati, and C. Silva. “Reproducibility using VisTrails,” *Implementing Reproducible Computational Research*, 33, 2014.

[4] S. B. Davidson, and J. Freire, “Provenance and scientific workflows: challenges and opportunities,” *Proceeding of the 2008 ACM SIGMOD international conference on Management of data.* ACM, 2008, pp. 1345-1350.

[5] J. L. R. Stevens, M. Elver, and J. A. Bednar, “An automated and reproducible workflow for running and analyzing neural simulations using Lancet and IPython Notebook,” *Frontiers in neuroinformatics*, vol. 7, p. 44, December 2013.

[6] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble, “The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud,” *Nucleic Acids Research*, vol. 41, no. W1, pp. W557–W561, 2013.

[7] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire, “noWorkflow: capturing and analyzing provenance of scripts,” *International Provenance and Annotation Workshop*. Springer International Publishing, 2014, pp. 71-83.

[8] T. McPhillips, T. Song, T.r Kolisnik, S. Aulenbach, K. Belhajjame, K. Bocinsky, "YesWorkflow: a user-oriented, language-independent tool for recovering workflow information from scripts." *arXiv preprint arXiv*:1502.02403. 2015 Feb 9.

[9] T. McPhillips, S. Bowers, K. Belhajjame, and B. Lud äscher, "Retrospective provenance without a runtime provenance recorder." *USENIX Workshop on Theory and Practice of Provenance*. July 2015.

[10] The VisTrails Project. http://www.vistrails.org. [Accessed 17 April 2017]

[11] S. Dey, K. Belhajjame, D. Koop, M. Raul, and B. Ludascher, "Linking ¨ prospective and retrospective provenance in scripts," *7th USENIX Workshop on the Theory and Practice of Provenance* (TaPP 15), 2015.

[12] K. Belhajjame, O. Corcho, D. Garijo, J. Zhao, P. Missier, D. Newman, R. Palma, S. Bechhofer, E. Garc´ ıa Cuesta, J. M. Gomez-P ´erez ´et al., "Workflow-centric research objects: First class citizens in scholarly discourse," *Proceedings of Workshop on the Semantic Publishing*, (SePublica 2012), 2012.

[13] Pymat Home. http://pymat.sourceforge.net [Accessed 17 April 2017]

[14] Mlabwrap Home. http://mlabwrap.sourceforge.net [Accessed 17 April 2017]

[15] Python interface to MATLAB. https://pypi.python.org/pypi/pymatlab [Accessed 17 April 2017]

[16] A. Sterian. "PyMat—an interface between Python and MATLAB." *Grand Valley State University, Allendale, MN*. 1999.

[17] YesWorkflow project site and README. https://github.com/yesworkflow-org/yw-prototypes, [Accessed 17 April 2017]

[18] S. B. Davidson, S. C. Boulakia, A. Eyal, B. Ludascher, T. M. ¨McPhillips, S. Bowers, M. K. Anand, and J. Freire. 'Provenance in Scientific Workflow Systems." *IEEE Data Eng. Bull.*, 30(4):44–50, 2007.

[19] J. Frew, D. Metzger, and P. Slaughter. "Automatic capture and reconstruction of computational provenance." *Concurrency and Computation: Practice and Experience*, 20(5):485–496, 2008.

[20] MATLAB http://www.mathworks.com/ [Accessed 17 April 2017]

[21] Y. Zhao, M. Wilde, and I. Foster. "Applying the virtual data provenance model." *International Provenance and Annotation Workshop (IPAW)*, 2006.

[22] Y. Dong, B. Wang. Mixed language programming with python and matlab. *Modern Electronic Technique*. *30*(14), 108-110, 2007.

[23] W. Aalst and K. Hee. Workflow Management: Models, Methods, and Systems. MIT Press, 2002.

[24] I. Altintas, O. Barney, and E. Jaeger-Frank. “Provenance collection support in the kepler scientific workflow system.” *International Provenance and Annotation Workshop.* Springer Berlin Heidelberg, pages 118–132, 2006.

[25] R. S. Barga and L. A. Digiampietri. “Automatic capture andefficient storage of escience experiment provenance.” *Concurrency and Computation: Practice and Experience,* 20(5):419–429, 2008.

[26] S. Bowers, T. McPhillips, and B. Ludaescher. “A provenance model for collection-oriented scientific workflows.” *Concurrency and Computation: Practice and Experience*, 20(5):519–529, 2008.

[27] R. Bose, I. Foster, and L. Moreau. "Report on the International Provenance and Annotation Workshop:(IPAW'06) 3-5 May 2006, Chicago." *ACM SIGMOD Record* 35.3 (2006): 51-53.

[28] S. Cohen, S. C. Boulakia, and S. B. Davidson. “Towards a model of provenance and user views in scientific workflows.” *International Workshop on Data Integration in the Life Sciences* Springer Berlin Heidelberg. (pp. 264-279).

[29] Y. L. Simmhan, B. Plale, and D. Gannon. "A survey of data provenance in e-science." *ACM Sigmod Record* 34(3):31–36, 2005.

[30] Y. L. Simmhan, B. Plale, and D. Gannon. “Karma2: Provenance management for data driven workflows.” *Web Services Research for Emerging Applications: Discoveries and Trends: Discoveries and Trends* :317, 2010.

[31] Y. L. Simmhan, B. Plale, D. Gannon, and S. Marru. "Performance evaluation of the karma provenance framework for scientific workflows." *International Provenance and Annotation Workshop.* Springer Berlin Heidelberg, 2006.

[32] J. Golbeck and J. Hendler. “A semantic web approach to tracking provenance in scientific workflows.” *Concurrency and Computation: Practice and Experience,* 20(5):431–439, 2008.

[33] P. Missier, S. Woodman, H. Hiden, and P. Watson. “Provenance and data differencing for workflow reproducibility analysis.” *Concurrency and Computation: Practice and Experience,* vol. 28, no. 4, pp. 995–1015, 2016.

[34] C. Bochner, R. Gude, A. Schreiber. “A Python library for provenance recording and querying.” *International Provenance and Annotation Workshop (IPAW)*. pp. 229–240, 2008.

[35] J. Kim, E. Deelman, Y. Gil, G. Mehta, V. Ratnaka. “Provenance trails in the wings/pegasus system.” *Concurrency and Computation: Practice and Experience*, 20(5), 587-597, 2008.

[36] M. R. Huq, P. M. Apers, A. Wombacher. “ProvenanceCurious: a tool to infer data provenance from scripts.” *Proceedings of the 16th International Conference on Extending Database Technology* (pp. 765-768). ACM. 2013

[37] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, B. Plale. “The open provenance model core specification (v1. 1).” *Future generation computer systems*, 27(6), 743-756. 2011.

[38] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, H. T. Vo. “Vistrails: Enabling interactive multiple-view visualizations.” *Visualization*, 2005. VIS 05. IEEE (pp. 135-142).

[39] J. Frew and R. Bose. “Earth system science workbench: A data management infrastructure for earth science products.” *Scientific and Statistical Database Management, 2001. SSDBM 2001. Proceedings. Thirteenth International Conference on.* IEEE, 2001.

[40] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C.T. Silva, H. T. Vo, "Managing the evolution of dataflows with VisTrails." *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*. IEEE, 2006.

[41] D. Koop, J. Freire, C. T. Silva. "Enabling reproducible science with VisTrails." *arXiv preprint arXiv*:1309.1784, 2013.